1. **Introduction**

In this lab, we will work with the PriorityQueue ADT, and with different implementations that have been introduced in lectures and in the textbook. We start by reviewing some relevant concepts about the PriorityQueue ADT. See a discussion about the complete binary trees and the heap at the end of this document.

The priority queue is a collection of elements represented by what we refer to as *entries*. An *entry* is a special type of object that associates two values, a *key* value and a *data* value; and it is specified as follows:

|  |
| --- |
| /\*\* ADT specification of Entry. An entry is a pair of values: one representing a key  (of type K) and one representing a data value (of type V) that is associated with  the particular key value.  \*\*/  public interface Entry<K, V> {  /\*\* Accesses the key of the entry.  @return reference to the key value of the entry.  \*\*/  K getKey();  /\*\* Accesses the value of the entry.  @return reference to the value in the entry.  \*\*/  V getValue();  } |

The keys are assumed *comparable* based on some *order relation*; which, for any particular instance, will be defined by an *associated comparator*. No assumption is imposed on the values that are associated with keys in a given map (the other component of each entry in the map). Based on that order relation of keys, *the smaller the value of the key, the higher is the priority of an entry*.

This type of collection is very useful on applications in which prioritized access is relevant. Common service oriented systems usually use priority queues when deciding what request to serve next whenever a large number of requests are waiting for attention by the system and some may have higher urgency than others; for example, network routers, operating systems, projectiles defense systems, communication systems, etc. In addition, several important algorithms use a priority queue as a fundamental data structure; some are: finding min cost spanning trees, finding shortest paths in a network, etc. They are also very useful in simulation of systems based on discrete events.

In this lab, we shall work with different implementations of the PriorityQueue ADT. What follows is its specification as seen in lectures.

|  |
| --- |
| public interface PriorityQueue<K, V> {  /\*\* The size of the collection…. \*\*/  int size();    /\*\* Returns true if empty; false, otherwise. \*\*/  boolean isEmpty();  /\*\* Add a new entry (key, value) to the queue.  @param key the key for the entry  @param value the value for the entry  @return Reference to the Entry object that is internally created for the pair key-value  @throw IllegalArgumentException if key is not valid for a particular instance  \*\*/  Entry<K, V> insert(K key, V value) throws IllegalArgumentException;  /\*\* Accesses entry in the collection having highest priority (or minimum value of the  key according to a particular order relation)  @return Reference to the entry with min key value, or null if empty.  \*\*/  Entry<K, V> min();  /\*\* Removes entry in the collection having highest priority (or minimum value of the  key according to a particular order relation)  @return Reference to the entry removed, or null if empty  \*\*/  Entry<K, V> removeMin();  } |

1. **About This Lab**

You have received a Java project that contains some partial implementations of PriorityQueue as specified in lectures and in the textbook. They are based on a hierarchy of classes to achieve a good level of code reusability on those different implementations. There are some minor differences from what is given in the book; some of them, just for the purpose of this lab and for testing purposes. These are:

1. A new version of priority queue is introduced; this is given as the Java interface DisplayablePriorityQueue. The only purpose of this is to allow display of the priority queue content. Notice that it is an interface that extends PriorityQueue interface, and that this new interface only specifies a new method: display(). The purpose of this method is to display the current content (entries) in the priority queue and hence allow one to see what is inside it. The order of how the elements are displayed, or structure of the displayed elements, will depend on particular final implementations. Therefore, different implementations may need to implement their own display method.
2. The AbstractPriorityQueue class now implements DisplayablePriorityQueue. Also, in that class, the comparator has been changed[[1]](#footnote-0) to EntryComparator, but it does essentially the same as in the implemented version in the textbook. An entry comparator is based on a comparator for the keys. From the point of view of any subclass of this abstract class, or from objects instantiated using those subclasses, nothing has changed from what we discussed in lectures.
3. The implementations based on lists are now based on objects of type List; in particular, they use ArrayList (What we studied in lectures is based on PositionalList.) And, in order to achieve more code reusability, we have introduced the abstract class: AbstractListPriorityQueue. That abstract class implements some methods that apply to all subclasses that will eventually extend it with the objective to provide a full implementation of PriorityQueue. There are three different implementations based on lists: UnsortedListPriorityQueue, SortedListPriorityQueue, and HeapListPriorityQueue. These three are similar to the corresponding implementations seen in lectures (there, using PositionalList for the first two and the ArrayList for the third one). You are expected to easily understand these new versions because we are using the same idea as discussed in lectures, now using an internal object of type List to store the entries in the priority queue. The third priority queue implementation that is based on an object of type List is the *heap-based* version as implemented in the Java class HeapListPriorityQueue. This basically follows the same approach that we studied in lectures. In all of them, another difference with what was studied in lectures is the intermediate abstract class AbstractListPriorityQueue.
4. Class HeapPriorityQueue<K, V> is another implementation of the PriorityQueue<K, V> ADT that is also based on the heap data structure. Notice that it has an internal field of type Heap<Entry<K, V>>. More details about the Heap are given next. This class is introduced here for the purpose of visualization of the tree structure when implementing the priority queue using the complete binary tree approach. With this class, we can take advantage of the operation to display the content of a tree as seen in earlier activites in the lab. It is useful for learning purposes but perhaps not as a real implementation of the priority queue, since it has more overhead than the HeapListPriorityQueue; although asymptotically the execution times should be the same.

To support its implementation, two other classes have been introduced to implement the *heap* as a particular type of BinaryTree, hence taking advantage of the abstract implementation of the BinaryTree. These are the two Java classes: CompleteBinaryTree, which is a subclass of AbstractBinaryTree; and the Heap class, which is a subclass of CompleteBinaryTree that implements a *min-heap* ADT (a complete binary tree satisfying the heap property - see below).

* 1. Class CompleteBinaryTree is a BinaryTree (as specified in lectures) which is complete. As it is the case for binary trees in the textbook, it works with positions. But, because of the peculiarities of the complete binary tree, we know that it can be efficiently represented using an array (or an ArrayList); therefore, here, positions are not nodes as were in class LinkedBinaryTree (as in the textbook). We implement Position using an internal class: CBTPosition<E> (complete binary tree position), and the internal array is declared as ArrayList<CBTPosition<E>>. Study the internal class CBTPosition<E> as partially included below. Such positions have two internal fields: element and index. The element is as usual (to hold the element at the position), but the index is needed here because, since the positions are stored in the internal list (array or index list), and also several operations of the BinaryTree require reference to a position, any position object used here needs to know where in the list it is located. That is precisely the purpose of that index.

|  |
| --- |
| protected static class CBTPosition<E> implements Position<E> {  private E element; // the element at this position  private int index; // index of its position in the array list  public CBTPosition(E element, int index) {  this.element = element;  this.index = index;  }  ... getters and setters....  } |

See for example the implementation of the parent(...) operation in class CompleteBinaryTree.

Also notice that this class has an add operation. It adds a new element to the tree, by just adding another node at the end of the tree, and which ends holding the element to be added. That only requires adding it to the end of the internal list (the ArrayList).

* 1. Class Heap basically has the same operations as the PriorityQueue ADT, but it does not work with entries, it works with values[[2]](#footnote-1) directly. The values are assumed to be comparable based on an order relation that is defined by a Comparator object. That is particularly important for the result of the min() and removeMin() operations. So, in this case, the Heap is a *complete binary tree* that holds data in its nodes and which satisfies the **heap property**: *that every position (different from the root) in the tree stores a value which is “greater or equal”[[3]](#footnote-2) to the value stored at its parent position*. Remember that such property guarantees that the value at the root position is always a min value. This is also called the *min-heap property*.

Now, if you go to the Heap class (which is a subclass of CompleteBinaryTree), there you will see some new public operations: min() and removeMin(). The first one accesses the element in the heap with minimum value and the second removes it the heap. Both return reference to that min value, or return null if the heap is empty. Also notice that the add operation is reimplemented in this class (operation inherited from class CompleteBinaryTree). It overrides the add operation in class CompleteBinaryTree, with the goal of properly placing the new element inside the heap tree (not just at the end of the complete binary tree) and hence guaranteeing that the internal structure continues satisfying the heap property.

The partial implementation of Heap class also includes a partial implementation of downHeap(...), and upHeap(...), which are auxiliary methods for the operations add(..) and removeMin() inside this class. The two internal methods, downHeap and upHeap, will make use, as needed by their respective algorithms, of the other internal method: swapPositionsInList(...). Notice that when a position is moved to another location, the internal index of the position needs to be modified accordingly.

|  |
| --- |
| /\*\*  \* Interchanges two position in the array.  \* @param r one of the position  \* @param c the other position  \*/  private void swapPositionsInList(CBTPosition<E> r, CBTPosition<E> c) {  int ir = r.getIndex();  int ic = c.getIndex();  // since positions change location, indexes are changed too  r.setIndex(ic);  c.setIndex(ir);    // swap content of location ir and ic in the arraylist  list.set(ir, list.set(ic, r)); // swaps elements at positions ir and ic in list    } |

1. **Exercises**

Please, import the partial project as an Eclipse project in your system. Notice that this partial implementation includes classes for trees. We will use them here too as described earlier. Also, in package testerClasses there are two particular tester classes that we will use in the following exercises. Those are: HeapTester1 and PriorityQueueTester1.

1. Go inside file Heap.java, which is in package heap. Look for the method downHeap (see idea for this algorithm in the discussion at the end of this document or in the textbook), in particular, for the comment stating that code is missing. Add the missing code. Once you add the correct code, if you run the class program HeapTester1, you should see the output shown in outputFile1.
2. Now, in package testerClasses, add a new comparator class named IntegerComparator2. That class should cause that if you run the same program again (HeapTester1), but changing the 1 for a 2 in line \*\*, the output will be as in file outputFile2. The heap now behaves as as max-heap.

If you take a look inside PriorityQueueTester1.java, you should see the following four lines.

|  |
| --- |
| DisplayablePriorityQueue<Integer, String> pq = new UnsortedListPriorityQueue<>(new IntegerComparator1()); //1  //DisplayablePriorityQueue<Integer, String> pq = new SortedListPriorityQueue<>(new IntegerComparator1()); //2  //DisplayablePriorityQueue<Integer, String> pq = new HeapListPriorityQueue<>(new IntegerComparator1()); //3  //DisplayablePriorityQueue<Integer, String> pq = new HeapPriorityQueue<>(new IntegerComparator1()); //4 |

In the following exercises, we will refer to them as Line 1, Line 2, Line 3, and Line 4; based on the number at the right side. Only one of those should be uncommented at any moment.

1. Run the class program PriorityQueueTester1. (Make sure that, of the above four lines, the only uncommented one is Line 1.) The program shows the following output:

|  |
| --- |
| PQ content after adding entry: key = 20 and value = twenty  [20, twenty]  PQ content after removing highest priority element [20, twenty]  EMPTY  Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0  at java.util.ArrayList.rangeCheck(ArrayList.java:653)  at java.util.ArrayList.remove(ArrayList.java:492)  at priorityQueue.AbstractListPriorityQueue.removeMin(AbstractListPriorityQueue.java:66)  at testerClasses.PriorityQueueTester1.removeMin(PriorityQueueTester1.java:65)  at testerClasses.PriorityQueueTester1.main(PriorityQueueTester1.java:27) |

Discover what the error is. Once you correct it, the output produced should be as in file outputFile3.

1. Change the IntegerComparator1 for IntegerComparator2 in class PriorityQueueTester1. Then run again this program. Study the output and compare with the output on the previous exercise. Why is that?
2. Remove the comment characters (//) from line 2 inside PriorityQueueTester1 (make sure that is the only one that is uncommented out of the four lines). Execute the program. You will see the following error message:

|  |
| --- |
| PQ content after adding entry: key = 20 and value = twenty  Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: -1, Size: 0  at java.util.ArrayList.rangeCheckForAdd(ArrayList.java:661)  at java.util.ArrayList.add(ArrayList.java:473)  at priorityQueue.SortedListPriorityQueue.insert(SortedListPriorityQueue.java:32)  at testerClasses.PriorityQueueTester1.add(PriorityQueueTester1.java:61)  at testerClasses.PriorityQueueTester1.main(PriorityQueueTester1.java:26) |

Discover what the error is and correct it. Run the program again. Output should be as in file outputFile4. Once that is done, repeat execution for each of the other three lines; one by one, uncomment one line while commenting the other three. Execute the program in each case. The results should be the same (as far as the order in which the elements are added and removed).

1. Run previous program again, but using IntegerComparator2 instead of IntegerComparator1. Explain the results.
2. Consider class program PriorityQueueTester2. It invokes static methods (displayArray and what), which are inside class TesterUtils. Inside method what, two lines are missing. The lines are meant for the program PriorityQueueTester2 to produce output as shown in file outputFile5. You cannot alter any other part of that class; just add the two correct lines.

1. Run the previous program again, but instead of using IntegerComparator1, use comparator IntegerComparator2. Explain the results.
2. Write a new tester similar to PriorityQueueTester2, and which uses the following array instead:

|  |
| --- |
| String[] arr = {"barrio", "pepe", "julia", "maria", "oliva", "meme", "parada", "baile", "enjendro",  "vagabundo", "nota", "tienda", "zapato", "caballo", "cafe", "diodo", "multiplica"}; |

Name it PriorityQueueTester3. The last three lines have to be exactly as in PriorityQueueTester2. Test with the different implementations of priority queue being considered here and with the following comparator:

|  |
| --- |
| public class StringComparator1 implements Comparator<String> {  public int compare(String s1, String s2) {  return s2.compareTo(s1);  }  } |

**Some General Discussion About Complete Binary Trees and the Heap**

Remember that a binary tree is a tree in which every node has at most two children. Children of a node are distinguished as *left child* and *right child*. So far we have seen an implementation of binary trees using a linked structure. However, such ADT can be implemented using an array. For this, we can assign an index value, i(v), to each node v in a binary tree t as follows:

|  |
| --- |
| i(t.root()) = 0  i(t.left(v)) = 2\*i(v)+1  i(t.right(v)) = 2\*i(v)+2 |

Under this indexing, we also have the following: if *v* is different from the root, then the following is true: *i(t.parent(v)) = floor((i(v)-1)/2)*. For example, the index of the parent of node having index 7 is floor(6/3) = 3. Notice that in Java, since i(v) is an integer, then floor((i(v)-1)/2) is the same as (i(v)-1)/2.

The above indexing of nodes in a binary tree suggests an alternative to store such a tree in memory using a 1-dimensional array. In that case, the index assigned to node v identifies the position of the array in which node v is represented. The following figure illustrates this. However, as you can see from the figure, this is not necessarily a memory efficient manner to represent a binary tree in general. Why? But the good news is that there is an important type of binary tree that can be efficiently represented using such array. That is the Complete Binary Tree, and it also happens that this type of tree is an excellent alternative to implement the priority queue efficiently; both, in terms of memory requirements as well as in terms of execution time of the algorithms for its operations.



**Complete Binary Tree**

An informal definition of a *complete binary tree* is as follows: Let t be a binary tree. If t has size 0 or 1, then it is a complete binary tree. If it has size > 1, and hence height h ≥ 1, it is complete if the following two conditions are satisfied:

1. all levels before the last level (levels 0, 1, ..., h-1) are full (these is the same as saying that for i in 0, 1, ..., h-1, that level i has 2i nodes (the maximum possible for a binary tree)), and
2. all its nodes on the last level occupy the left-most positions in that level. More formally, this condition establishes that if v is a node at the last level, then all nodes u at the previous level to the left of parent of v, if any such node, must have two children, and if v is the right child of its parent, then it must have a sibling node in the tree (the left child of its parent in that case).

The following figure shows some examples.

|  |  |
| --- | --- |
| Complete Binary Trees | Binary Trees that are not Complete |
|  | Violates condition 1 |
|  | Violates condition 2 |

A good property of complete binary trees is the following:

|  |
| --- |
| A binary tree t is complete if, and only if, for every node v in t, the index i(v) computed as described earlier satisfies: 0 ≤ i(v) < t.size(). |

This implies that complete binary trees can be efficiently represented using 1-dimensional arrays, in which node v corresponds to position i(v) in the array.

**Heap (min-heap)**

A heap is a complete binary tree t holding elements that satisfy some order relation and the following is true for every node v in t different from the root: *element at parent of v is “≤” element in v* (where the “≤” is based on the order relation that is satisfied by the elements). In our case, as we have seen, that order relation is established by a *comparator* object. That is, if cmp is that comparator object, then cmp.compare(t.parent(v).getElement(), v.getElement()) <= 0. (Note: this definition really corresponds to min-heap; similarly, we can have a max-heap.)

**Implementation of the Priority Queue Using a Heap**

We can use a heap to efficiently implement a priority queue. In that case, elements in the tree are entries as seen in lectures and in the textbook, and hence the above inequality is based on the keys in those entries. For the following discussion, we assume that what we want to implement is a Heap<E> ADT. The following are the three key operations:

1. void add(E e): adds a new element to the heap. After the new element is added, the heap property will continue being satisfied.
2. E min(): returns reference to minimum element (based on the comparator).
3. E removeMin(): Removes minimum element from the heap. After the operation is completed, the internal structure must continue being a heap. It returns reference to the deleted element.

The last two operations return null for the case of an empty heap.

Since the heap is a complete binary tree, we implement it using an array-like structure, in this case, using an ArrayList<E>. However, the algorithms are better visualized and understood using the tree.

What to do when inserting a new element in the heap? The idea here is to add a “new node” (which is to contain the new element to be added) as the “next possible leaf” in the tree (for it to continue being a complete binary tree), and to reorganize the data in a way that the tree continues having the property of being a min-heap. This operation begins with the assumption that the tree is a min-heap (before adding the new node). The algorithm to reorganize the data in the tree is based on comparing the element in the new node, with the element in its parent node (if any) and switch elements if necessary. Assume that **e** is the new element being inserted to the heap. Also assume that **current** is the index value of the position in **h** where the new node is located in its way upward in the tree until reaching its final position. In that “moving upward”, once the final position is reached, the method guarantees that the tree is a min-heap again, with one new member. Initially, **current** is the last position of the array (the position of the “new node”).

For the following discussion assume that list is the internal ArrayList object holding the elements of the heap and organized as such (satisfying being a heap as per the definition given). What the algorithm does is first adds the new element at the end of list (list.add(e)). Then, since this new element added to the heap may alter the heap property, we might need to reorganize the list (the tree). For that, we just need to make sure that the path from that new position all the way to the root of the tree is ordered, from top to bottom, in increasing order. Luckily, we don’t need to worry about other parts of the heap; only that path. To do that we just need to move the new element upward along that path (as we did on sorted lists) until it is inserted in the right location for that path to be ordered as said before. This operation is usually referred to as upHeap operation. The following is a recursive algorithm for this operation in Java-like notation. Since the new element is originally added at position list.size()-1, then the algorithm shall be invoked as: upHeap(list.size()-1). Assume this is an internal private non-static method in the class....

|  |
| --- |
| **void** upHeap(int r) {  **if** (r > 0) {  int parentIndex = (r-1)/2  **if** (cmp.compare(list.get(r), list.get(parentIndex)) < 0)   1. swap elements in positions r and parentIndex in list 2. upHeap(parentIndex)   }  } |

One can easily come with an iterative version of this algorithm. Notice that at the most, the new element will be moved upward all the way to the root of the tree; this is the case when the new element becomes the new “min value” in the heap.

Similarly, when removing an element (**removeMin()**) from the heap, the element to remove is the one at the root of the tree (position 0 in list). Hence the tree needs to be reorganized to continue being a heap, but with one node (and one element) less. The general idea of the algorithm is as follows. This is usually referred to as the downHeap operation.

1. copy the element in root to a safe place (**etr**). (This is the element to return at the end.)
2. remove the last leaf from the tree (the only node that can be physically removed in order for the tree to continue being a complete binary tree)
3. if the tree is still not empty:
   1. copy the element in the node being removed to the root node (Notice that we now have a tree that is complete, but may not be a heap. However, the good part is that both, the left and the right subtrees of that root node are heaps.)
   2. move that element downward in the tree (by switching with its minimum child (if any, and if the element of the minimum child is “less than” the element being moved downward) as needed until its correct position is reached for the tree to once again become a min-heap with one element (one node) less.

When moving the current node (initially the root) downward: you must do the following wherever needed in you algorithm (if current is not external; which happens if it has left child (See WHY?)):

1. int lci = 2\*current + 1; // left child index - node exists in the represented tree if lci < list.size()

int rci = lci+1; // right child index - node exists in the represented tree if rci < list.size()

int mci = lci; // min child index - initially assumed to be the left child...

1. if (… current has right child… )

if (cmp.compare(list.get(rci), list.get(lci)) < 0)

mci = rci;

1. if (cmp.compare(list.get(mci), list.get(current)) < 0) // element in current needs to move further down

list.set(current, list.set(mci, list.get(current))) /// this does the swap in the array list....

current = mci // to continue the process if needed

This can also be implemented recursively or iteratively *while current is not external* and the *element in current still needs to be moved downward*. Notice that at most, the element being moved downward will end in a leaf of the complete binary tree.

As you can see, these algorithms have execution time *O(height of the tree)* which *O(log n)*.

1. FIN …

1. When compared to the implementations in the textbook. [↑](#footnote-ref-0)
2. If used to implement a PriorityQueue as specified, then those values would be entries. [↑](#footnote-ref-1)
3. As per an order relation defined by an associated Comparator object. [↑](#footnote-ref-2)